# Semantic Mutation Testing for Multi-Agent Systems

Zhan Huang and Rob Alexander

Department of Computer Science, University of York, York, United Kingdom
{zhan.huang,robert.alexander}@cs.york.ac.uk

**Abstract.** This paper introduces semantic mutation testing (SMT) into multi-agent systems. SMT is a test assessment technique that makes changes to the interpretation of a program and then examines whether a given test set has the ability to detect each change to the original interpretation. These changes represent possible misunderstandings of how the program is interpreted. SMT is also a technique for assessing the robustness of a program to semantic changes. This paper applies SMT to three rule-based agent programming languages, namely Jason, GOAL and 2APL, provides several contexts in which SMT for these languages is useful, and proposes three sets of semantic mutation operators (i.e., rules to make semantic changes) for these languages respectively, and a set of semantic mutation operator classes for rule-based agent languages. This paper then shows, through preliminary evaluation of our semantic mutation operators for Jason, that SMT has some potential to assess tests and program robustness.

**Keywords:** Semantic Mutation Testing, Agent Programming Languages, Cognitive Agents

## 1 Introduction

Testing multi-agent systems (MASs) is difficult because MASs may have some properties such as autonomy and non-determinism, and they may be based on models such as BDI which are quite different to ordinary imperative programming. There are many test techniques for MASs, most of which attempt to address these difficulties by adapting existing test techniques to the properties and models of MASs [9, 13]. For instance, SUnit is a unit-testing framework for MASs that extends JUnit [17].

Some test techniques for MASs introduce traditional mutation testing, which is a powerful technique for assessing the adequacy of test sets. In a nutshell, traditional mutation testing makes small changes to a program and then examines whether a given test set has the ability to detect each change to the original program. These changes represent potential small slips. Work on traditional mutation testing for MASs includes [1, 10, 14–16].

In this paper, we apply an alternative approach to mutation testing, namely semantic mutation testing (SMT) [5], to MASs. Rather than changing the program, SMT changes the semantics of the language in which the program is written. In other words, it makes changes to the interpretation of the program. These changes represent

possible misunderstandings of how the program is interpreted. Therefore, SMT assesses a test set by examining whether it has the ability to detect each change to the original interpretation of the program.

SMT can be used not only to assess tests, but also to assess the robustness of a program to semantic changes: Given a program, if a change to its interpretation cannot be detected by a trusted test set, the program is considered to be robust to this change.

This paper makes several contributions. First, it applies SMT to three rule-based agent programming languages, namely Jason, GOAL and 2APL. Second, it provides several contexts (scenarios) in which SMT for these languages is useful. Third, it proposes three sets of semantic mutation operators (i.e., rules to make semantic changes) for these languages respectively, and a broader set of semantic mutation operator classes (that serve as a guide to derivation of semantic mutation operators) for rule-based agent languages. Finally, it presents a preliminary evaluation of the semantic mutation operators for Jason, which shows some potential of SMT to assess tests and program robustness.

The remainder of this paper is structured as follows: Section 2 describes two types of mutation testing – traditional mutation testing and semantic mutation testing. Section 3 describes SMT for Jason, GOAL and 2APL by showing several contexts in which it is useful and the source of semantic changes required to apply SMT in each context. Section 4 proposes sets of semantic mutation operators for these languages and a set of semantic mutation operator classes for rule-based agent languages. Section 5 evaluates the Jason semantic mutation operators. Section 6 summarizes our work and suggests where this work could go in the future.

## 2 Mutation Testing

### 2.1 Traditional Mutation Testing

Traditional mutation testing is a test assessment technique that generates modified versions of a program and then examines whether a given test set has the ability to detect the modifications to the original program. Each modified program is called a *mutant*, which represents a realistic small fault in the program. Mutant generation is guided by a set of rules called *mutation operators*. For instance, Figure 1(a) shows a piece of a program and Figure 1(b) – 1(f) show five mutants generated as the result of the application of a single mutation operator called *Relational Operator Replacement*, which replaces one of the relational operators ($<, \leq, >, \geq, =, \neq$) with another operator.

After mutant generation, the original program and each mutant are executed against all tests in the test set. For a mutant, if its resultant behaviour differs from the behaviour of the original program on some test, the mutant will be marked as *killed*, which indicates that the corresponding modification can be detected by the test set. Therefore, the fault detection ability of the test set can be assessed by the *mutant kill rate* – the ratio of the killed mutants to all generated mutants: the higher the ratio is, the more adequate the test set is. In the example shown in Figure 1, a test set consisting of a single test in which the input is *x=3, y=5* cannot kill the mutants shown in Figure 1(b) and 1(f) because on that test these two *live* mutants result in the same

behaviour as the original program (i.e., *return a*). Therefore, the mutant kill rate is 3/5. According to this result we can enhance the test set by adding a test in which the input is *x=4, y=4* and another test in which the input is *x=4, y=3* in order to kill these two live mutants respectively and get a higher mutant kill rate (the highest kill rate is 1, as this example shows).

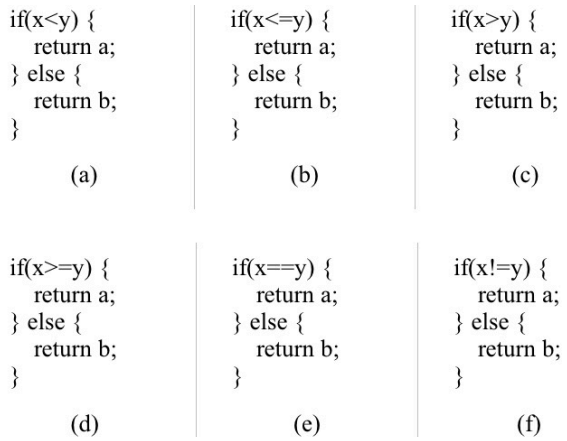


**Fig. 1.** An example of traditional mutation testing

Many studies provide evidence that traditional mutation testing is a powerful test assessment technique, so it is often used to assess other test techniques [2, 12]. However, the mutation operators used to guide mutant generation may lead to a large number of mutants because a single mutation operator has to be applied to each relevant point in the program and a single mutant only contains a modification to a single relevant point (as shown in Figure 1). This makes comparing the behaviour of the original program and that of each mutant on each test is computationally expensive.

Another problem is that traditional mutation testing unpredictably produces equivalent mutants – alternatives to the original program that are not representative of faulty versions, in that their behaviour is no different from the original in any way that matters for the correctness of the program. Thus, no reasonable test set can detect the modifications they contain. Equivalent mutants must therefore be excluded from test assessment (i.e., the calculation of the mutant kill rate). The exclusion of equivalent mutants requires much extra manual work although this process may be partially automated.

### 2.2 Semantic Mutation Testing

Clark et al. [5] propose semantic mutation testing (SMT) and extend the definition of mutation testing as follows: suppose N represents a program and L represents the semantics of the language in which the program is written (so L determines how N is interpreted), the pair (N, L) determines the program's behaviour. Traditional mutation testing generates modified versions of the program namely $N \rightarrow (N_1, N_2, \ldots, N_k)$

while SMT generates different interpretations of the same program namely L → ($L_1$, $L_2$, …, $L_k$). For SMT, $L_1$, $L_2$, …, $L_k$ represent *semantic mutants*, the generation of which is guided by a set of *semantic mutation operators*. For instance, Figure 2 shows a piece of a program, a semantic mutant (i.e., a different interpretation of this program) is generated by the application of a single semantic mutation operator that causes the *if* keyword to be used for mutual exclusion (i.e., when an *if* is directly followed by another *if*, the second *if* statement is interpreted the same as an *else-if* statement).

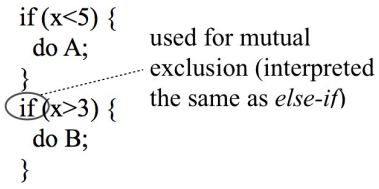


**Fig. 2.** An example of semantic mutation testing

SMT assesses a test set in a similar way as traditional mutation testing – comparing the system behaviour each semantic mutant results in with that the original interpretation results in so as to detect the killed mutants. In the example shown in Figure 2, a test set consisting of a single test in which the input is *x=2* cannot kill the semantic mutant because on that test the mutant results in the same behavior as the original interpretation (i.e., only *do A*). Therefore, the mutant kill rate is 0/1 = 0. We can enhance this test set by adding another test in which the input is *x=4* in order to kill the live mutant.

SMT is a useful test assessment technique because it can simulate a different class of faults than traditional mutation testing – possible misunderstandings of how the program is interpreted rather than small slips. Although semantic changes can be simulated by changes to the program, SMT often requires higher order (traditional) mutation[1] to simulate a semantic change, and empirical studies (e.g., [11]) show that some higher order mutants are harder to kill than first-order mutants. In addition, [5] show that SMT has potential to capture some faults that cannot be captured by traditional mutation testing.

SMT can be used not only to assess tests, but also to assess the robustness of a program to semantic changes. Given a semantic mutant, if it cannot be killed by a trusted

[1] Higher order mutation generates a higher order mutant by making more than one change to the program (these changes may form a subtle fault that is hard to detect). In contrast, most traditional mutation is first order, which generates a first order mutant by making only a single and simple change to the program.

test set[2], it will be considered as "equivalent"[3], which indicates that the program is robust to the corresponding semantic change, otherwise the program may need to be improved to resist this change. In the example shown in Figure 2, if the program is required to be robust to the semantic change, it can be modified to ensure that only one branch is executed in any case.

SMT has another difference to traditional mutation testing: it generates far fewer mutants because a single semantic mutation operator only leads to a single semantic mutant[4], namely a different interpretation of the same program (as shown in Figure 2), while a single traditional mutation operator may lead to many mutants each of which contains a modification to a single relevant point in the program (as shown in Figure 1). This makes SMT less computationally costly.

We know that SMT makes semantic changes for assessing tests or program robustness. For a particular language, which semantic changes should be made by SMT are context-dependent. For instance, to assess tests for a program written by a novice programmer, semantic changes to be made can be derived from common novices' misunderstandings. To assess the portability of a program between different versions of the interpreter, semantic changes to be made can be derived from the differences between these versions.

## 3 Semantic Mutation Testing for Jason, GOAL and 2APL

We investigate semantic mutation testing for MASs by first applying it to three rule-based programming languages for cognitive agents, namely Jason, GOAL and 2APL. These languages have similar semantics – an agent deliberates in a cyclic process in which it selects and executes rules according to and affecting its mental states. They also have similar constructs to implement such agents such as beliefs, goals and rules. The details of these languages can be found in [4, 6, 8] and are not provided here.

From Section 2 we know that for a particular language, the semantic changes that can most usefully be made by SMT is context-dependent. In the remainder of this section we provide several contexts in which SMT for the chosen agent languages is useful – migration between languages, evolution of languages, common misunderstandings, and ambiguity of informal semantics. We also show the source of semantic changes required to apply SMT in each context.

---

[2] A trusted test set is the one that is considered as "good enough" for the requirement. It doesn't need to be the full test set that is usually impractical; instead it can choose not to cover some aspects or to tolerate some errors.

[3] Here the term "equivalent" is different to the one used in the context of test assessment, in which a mutant is equivalent only if there exist no tests that can kill this mutant. In the context of robustness assessment, a mutant is equivalent if only the trusted test set cannot kill it.

[4] This rule can be relaxed, namely mutating the semantics of only parts of the program instead of mutating the semantics of the whole program. This is useful e.g., when the program is developed by several people.

### 3.1 Migration between Languages

When a programmer migrates a program from one language to another, or simply starts to write a new program in a new (to him or her) language, he or she may have misunderstandings that come from the semantic differences between the new language and the old one(s) he or she ever used. Therefore, in order for SMT to simulate such misunderstandings, we should first find out their source, namely the semantic differences, by comparison between Jason, GOAL and 2APL. Since these languages each have large semantic size and distinctive features, we use the following strategies to guide the derivation of the semantic differences.

- Dividing the semantics of each of these languages into five aspects, as shown in Table 1. We do this because first of all, it provides a guide to derivation of semantic differences. Second, we focus on examining four aspects of the semantics, namely deliberation step order, rule selection, rule execution, and mental state query and update, which are important and common to rule-based agent languages. We also roughly examine other aspects of the semantics in order for completeness. Finally, it is reasonable that common aspects of the semantics are more likely to cause misunderstandings than distinctive aspects in the context of migration, because distinctive aspects are usually supported by distinctive constructs that a programmer would normally take time to learn.
- Focusing on semantic differences between similar constructs. As [5] suggests, such differences easily cause misunderstandings because when migrating a program a programmer may just copy the same or similar constructs without careful examination of their semantics given by the new language.
- Examining both formal and informal semantics of these languages. We start with examining the formal semantics because they can be directly compared. We also verify those that are informally defined through programming and examination of the interpreter source code.
- Focusing on the default options of the interpreter. The interpreters for these languages are customizable, for instance, the Jason agent architecture can be customized by inheritance of the Java class that implements the default agent architecture; the GOAL rule selection order can be customized in the GOAL agent description. We think default options are more likely to cause misunderstandings in the context of migration because if a programmer customizes an element it suggests he or she is familiar with its semantics.

**Table 1.** The aspects of the semantics of Jason, GOAL and 2APL (those marked with an asterisk are the ones we focus on)

| ID | Aspect | Description |
|---|---|---|
| 1 | Deliberation step order* | Each deliberation cycle consists of a sequence of steps, e.g., rule selection → rule execution is a two-step sub-sequence. |
| 2 | Rule selection* | Rule selection is an important deliberation step in which one or several rules are chosen to be new execution candidates. |
| 3 | Rule execution* | Rule execution is an important deliberation step in which one or several execution candidates are executed. |
| 4 | Mental state query and update* | Mental states (i.e., beliefs and goals) can be queried in some deliberation steps such as rule selection and updated by execution of rules. |
| 5 | Other | Other aspects of the semantics not listed above. |

We present in Table 2 the semantic differences we found between Jason, GOAL and 2APL. These form the source of semantic changes required to apply SMT in the context of migration between these languages.

Difference 1 comes from the order of two important deliberation steps, namely rule selection and rule execution. A Jason agent first selects a rule to be a new execution candidate and then executes an execution candidate. A GOAL agent processes its *modules* one by one, in each module it first selects and executes event rules and then selects and executes an action rule (both event and action rules are defined in the module being processed). A 2APL agent first selects action rules to be new execution candidates, and then executes all execution candidates, next selects an external event rule, an internal event rule and a message event rule to be new execution candidates.

Difference 2 comes from the rule selection deliberation step. Jason, GOAL and 2APL differ in two aspects of this step, namely the rule selection condition and the default rule selection order. For the rule selection condition, a Jason or 2APL rule can be selected to be a new execution candidate if both its trigger condition and guard condition get satisfied ("applicable"), while a GOAL rule can be selected if it is applicable and the pre-condition of its first action gets satisfied ("enabled"). For the default rule selection order, Jason rules are selected in linear order (i.e., rules are examined in the order they appear in the agent description, and the first applicable rule is selected), GOAL action rules are selected in linear order while GOAL event rules are selected in "linearall" order (i.e., rules are examined in the order they appear in the agent description, and all enabled rules are selected), 2APL action rules are selected in "linearall" order while 2APL event rules of each type (external, internal, message) are selected in linear order.

Difference 3 comes from the rule execution deliberation step. In this step a Jason agent executes a single action in a single execution candidate, a GOAL agent executes all actions in each selected event rule and each selected action rule[5], a 2APL agent executes a single action in each execution candidate.

[5] Unlike Jason and 2APL, a GOAL agent has no intention set or similar structure, so a GOAL rule is immediately attempted to completely execute once selected.

**Table 2.** Semantic differences between Jason, GOAL and 2APL

| ID | Source | Jason | GOAL | 2APL |
|---|---|---|---|---|
| 1 | The order of rule selection and rule execution | select a rule → execute a rule | (select and execute event rules → select and execute an action rule) x Number_of_Modules | select action rules → execute rules → select an external event rule → select an internal event rules → select a message event rule |
| 2 | Rule selection | • applicable<br>• linear | • enabled<br>• linear (action rules) and linearall (event rules) | • applicable<br>• linear (event rules) and linear-all (action rules) |
| 3 | Rule execution | • one rule/cycle<br>• one action/rule | • one rule/cycle (action rules) and all rules/cycle (event rules)<br>• all actions/rule | • all rules/cycle<br>• one action/rule |
| 4 | Belief query | linear | random | linear |
| 5 | Belief addition | start | end | end |
| 6 | Goal query | $E \rightarrow I$; linear | random | linear |
| 7 | Goal addition | end of $E$ | end | start or end |
| 8 | Goal deletion | delete the event or intention that relates to the goal $\varphi$ | delete all super-goals of the goal $\varphi$ | delete the goal $\varphi$, all sub-goals of $\varphi$ or all super-goals of $\varphi$ |
| 9 | Goal type | procedural | declarative | declarative |
| 10 | Goal commitment strategy | no | blind | blind |

Difference 4 comes from the belief query. In a Jason or 2APL agent, beliefs are queried in linear order (i.e., beliefs are examined in the order they are stored in the belief base, and the first matched belief is returned). In a GOAL agent, beliefs are queried in random order (i.e., beliefs are randomly accessed, and the first matched belief is returned).

Difference 5 comes from the belief addition. In a Jason agent, a new belief is added to the start of the belief base. In a GOAL or 2APL agent a new belief is added to the end of the belief base.

Difference 6 comes from the goal query. For a Jason agent, since it keeps implicit goals or desires in goal type events and goal type intentions instead of keeping explic-it goals, it queries a goal by first examining its event base then its intention set, in

each of which it follows linear query order. In a GOAL agent, goals are queried in random order. In a 2APL agent, goals are queried in linear order.

Difference 7 comes from the goal addition. In a Jason or GOAL agent, a new goal is added to the end of the event or goal base. In a 2APL agent, a new goal is added to the start or the end of the goal base according to the relevant agent description (i.e., *adopta* or *adoptz*).

Difference 8 comes from the goal deletion. Given a goal $\varphi$ to be deleted, a Jason agent deletes the event or intention that relates to $\varphi$, a GOAL agent deletes all goals that have $\varphi$ as a logical sub-goal, a 2APL agent deletes $\varphi$, all goals that are a logical sub-goal of $\varphi$, or all goals that have $\varphi$ as a logical sub-goal according to the relevant agent description (i.e., *dropgoal*, *dropsubgoal* or *dropsupergoal*).

Difference 9 comes from the goal type. Jason adopts procedural goals – goals that only serve as triggers of procedures although it supports declarative goal patterns. GOAL and 2APL adopt declarative goals – goals that also represent states of affairs to achieve.

Difference 10 comes from the goal commitment strategy. Jason doesn't adopt any goal commitment strategy (i.e., a goal is just dropped once its associated intention is removed as the result of completion or failure) although it supports various commitment strategy patterns. GOAL and 2APL adopt blind goal commitment strategy, which requires a goal is pursued until it is achieved or declaratively dropped.

### 3.2 Evolution of Languages

When a programmer moves a program from a language to its successor, he or she may have misunderstandings that come from the semantic evolution. Another scenario is that a programmer may want to examine whether a program written in a language is compatible with a newer version of this language. To derive semantic changes required to apply SMT in these scenarios, we should first find out their source, namely the semantic differences between these languages/versions. We take 2APL and 3APL [7] as an example. 2APL is a successor of 3APL that modifies and extends 3APL. Table 3 shows some semantic differences between them. We explain these differences as follows.

**Table 3.** Semantics differences between 2APL and 3APL

| ID | Source | 2APL | 3APL |
|---|---|---|---|
| 1 | PR-rules | plan repair | plan revision |
| 2 | The order of rule selection and rule execution | see Table 2 | select an action rule → select a PR-rule → execute a rule |
| 3 | Action rule selection | linearall | linear |
| 4 | Rule execution | all rules/cycle | one rule/cycle |

Difference 1 comes from the PR-rules. In 2APL, the abbreviation "PR" means "plan repair", a PR-rule (i.e. an internal event rule) is selected only when a relevant plan fails. In 3APL, "PR" means "plan revision", a PR-rule is selected when matching some plan.

Difference 2 comes from the order of rule selection and rule execution deliberation steps. The order adopted by a 2APL agent has been described in Sub-section 3.1. In contrast, a 3APL agent selects an action rule then a PR-rule to be new execution candidates then executes an execution candidate.

Difference 3 comes from the action rule selection order. As described in Sub-section 3.1, 2APL action rules are selected in "linearall" order. In contrast, 3APL action rules are selected in linear order.

Difference 4 comes from the rule execution deliberation step. As described in Sub-section 3.1, a 2APL agent executes all execution candidates in a deliberation cycle. In contrast, a 3APL agent executes a single execution candidate.

### 3.3 Common Misunderstandings

A programmer may have misunderstandings that are common to a particular group of people he or she belongs to. Such misunderstandings can be identified by analysis of these people's common mistakes or faults. We take GOAL as an example: Table 4 shows some possible misunderstandings of the GOAL's semantics that are derived from some common faults made by GOAL novice programmers [18]. We explain these misunderstandings as follows.

**Table 4.** Possible novice programmers' misunderstandings of GOAL

| ID | Fault | Possible Misunderstanding |
|---|---|---|
| 1 | Wrong rule order | By default rules are selected in another available order. |
| 2 | A single rule including two user-defined actions | A rule can have more than one user-defined action. |
| 3 | Using "if then" instead of "forall do" | Existential quantification can be used for universal quantification. |

Possible misunderstanding 1 comes from the fault of the wrong rule order. If a programmer makes this fault in the GOAL agent description, he or she may have the misunderstanding that rules are selected in another available order[6] by default, e.g., action rules are selected in "linearall" order rather than linear order.

Possible misunderstanding 2 comes from the fault of a single rule including two user-defined actions. If a programmer makes this fault, he or she may have the misunderstanding that this is allowed like other agent languages.

Possible misunderstanding 3 comes from the fault of using "if then" instead of "forall do". If a programmer makes this fault, he or she may have the misunderstanding that existential quantification can be used for universal quantification.

[6] GOAL supports four available rule evaluation orders: linear, linearall, random and randomall.

### 3.4 Ambiguity of Informal Semantics

A programmer may have misunderstandings of the semantics that are not precisely or formally defined. For instance, [3] gives two examples of such misunderstandings of Jason as shown in Table 5. We explain these misunderstandings as follows.

**Table 5.** Possible misunderstandings due to Jason's informal semantics

| ID | Source | Possible Misunderstanding |
|---|---|---|
| 1 | Goal deletion event | "when an intention fails" → "when an intention is removed" |
| 2 | Test goal addition event | "when a test goal action fails" → "when a test goal action is executed" |

Possible misunderstanding 1 comes from the goal deletion event (*-!e* or *-?e*). A goal deletion event is generated when an intention with the corresponding goal achievement event (*+!e* or *+?e*) fails. A programmer may have the misunderstanding that this event is generated when this intention is removed as the result of completion or failure.

Possible misunderstanding 2 comes from the test goal addition event (*+?e*). A test goal addition event is generated when the corresponding test goal action (*?e*) fails. A programmer may have the misunderstanding that this event is generated when this action is executed, which is similar to the achievement goal addition event (*+!e*).

### 3.5 Discussion

SMT for Jason, GOAL and 2APL is of particular interest in the contexts discussed above considering:

- These languages are similar. As mentioned above they have similar semantics and constructs. Subtle semantic differences between similar constructs easily cause misunderstandings.
- These languages have elements that are allowed to customize. By mutating the semantics to represent different customizations it is possible to explore the robustness of a program.

## 4 Semantic Mutation Operators for Jason, GOAL and 2APL

According to our derived sources of semantic changes required to apply SMT in different contexts, we derive three respective sets of semantic mutation operators for Jason, GOAL and 2APL as shown in Table 6 – 8. Due to space limitations we don't explain each semantic mutation operator in details.

We observe that most of these operators act on the four aspects of the semantics we focus on, namely deliberation step order, rule selection, rule execution and mental state query and update (see Table 1). By further analysis we derive a set of semantic mutation operator classes for rule-based agent languages as shown in Table 9. These

classes provide a guide to derivation of semantic mutation operators for these languages.

**Table 6.** Semantic mutation operators for Jason

| ID | Semantic Mutation Operator | Description |
|---|---|---|
| 1 | Rule selection order change (RSO) | linear → linearall |
| 2 | Rule execution strategy change (RES) | one rule/cycle → all rules/cycle |
| 3 | Rule execution strategy change 2 (RES2) | interleaved execution of rules → non-interleaved execution of rules |
| 4 | Belief query order change (BQO) | linear → random |
| 5 | Belief addition position change (BAP) | start → end |
| 6 | Goal query order change (GQO) | linear → random |
| 7 | Goal addition position change (GAP) | end → start |
| 8 | Goal deletion event semantics change (GDES) | "when a plan fails" → "when a plan is removed" |
| 9 | Test goal achievement event semantics change (TGAES) | "when a test goal action fails" → "when a test goal action is executed" |

**Table 7.** Semantic mutation operators for GOAL

| ID | Semantic Mutation Operator | Description |
|---|---|---|
| 1 | Rule selection and execution order change (RSEO) | select and execute event rules then an action rule → select and execute an action rule then event rules |
| 2 | Rule selection condition change (RSC) | enabled → applicable |
| 3 | Rule selection order change (RSO) | change between linear, linearall, random and randomall |
| 4 | Belief query order change (BQO) | random → linear |
| 5 | Belief addition position change (BAP) | end → start |
| 6 | Goal query order change (GQO) | random → linear |
| 7 | Goal addition position change (GAP) | end → start |
| 8 | Goal deletion strategy change (GDS) | delete all super-goals of $\varphi$ → delete only $\varphi$ or delete all sub-goals of $\varphi$ |
| 9 | The maximum number of user-defined actions change (MNUA) | 1 → more than 1 |
| 10 | Quantification type change (QT) | make existential quantification ("if then") used for universal quantification ("forall do") |

**Table 8.** Semantic mutation operators for 2APL

| ID | Semantic Mutation Operator | Description |
|---|---|---|
| 1 | Rule selection and execution order change (RSEO) | change the original order "select action rules → execute rules → select event rules" to "select action rules → select event rules → execute rules" or "select event rules → select action rules → execute rules" |
| 2 | Rule selection condition change (RSC) | applicable → enabled |
| 3 | Rule selection order change (RSO) | change between linear and linearall |
| 4 | Rule execution strategy change (RES) | all rules/cycle → one rule/cycle |
| 5 | Belief query order change (BQO) | linear → random |
| 6 | Belief addition position change (BAP) | end → start |
| 7 | Goal query order change (GQO) | linear → random |
| 8 | PR-rule semantics change (PRRS) | plan repair → plan revision |

**Table 9.** Semantic mutation operator classes for rule-based agent languages

| ID | Semantic Mutation Operator Class |
|---|---|
| 1 | Rule selection and execution order change |
| 2 | Rule selection condition change |
| 3 | Rule selection order change |
| 4 | Rule execution strategy change |
| 5 | Mental state query order change |
| 6 | Mental state addition position change |
| 7 | Mental state deletion strategy change |
| 8 | Other change |

## 5 Evaluation of Semantic Mutation Operators for Jason

We have implemented our derived semantic mutation operators for Jason (as shown in Table 6) by modifying the source code of the Jason interpreter. Here we use two Jason projects in a preliminary evaluation of these operators, in order to assess the potential of SMT to assess tests and program robustness.

The Jason projects we chose are two of the examples released with the Jason interpreter. The first project is a simple one called *Domestic Robot* (DR), in which a domestic robot gets beer from the fridge and then serves its owner the beer until the owner reaches a certain limit of drinking. The robot will ask a supermarket to deliver beer when the fridge is empty. The second project is a relatively complex one called *Gold Miners* (the 2nd version, "GM II"), in which two teams of gold-mining agents compete against each other to retrieve as many pieces of gold scatters as possible in a grid-like territory, finding suitable paths to then take the retrieved gold to a depot.

We use two sets of randomly generated tests to test these Jason projects respectively (40 tests for DR and 102 tests for GM II). Each test is a starting configuration of the Jason project, which is represented by a set of parameters extracted from the agent description and the environment description such as the limit of drinking and the map size.

We run each Jason project under the original interpreter and each modified version of the interpreter (that implements a semantic mutation operator) against the corresponding test set, after which we collect and analyze the SMT results. We present the final results in Table 10.

**Table 10.** Results of semantic mutation testing

| Semantic Mutation Operator | Resultant Mutant of DR | Resultant Mutant of GM II |
|---|---|---|
| RSO | NE | K |
| RES | E | E |
| RES2 | NE | K |
| BQO | E | E |
| BAP | E | NE |
| GQO | N/A | E |
| GAP | E | E |
| GDES | K | K |
| TGAES | K | N/A |

As is normal for SMT, a semantic mutation operator here leads to a single semantic mutant if the interpretation of the Jason project involves the relevant semantics; otherwise the operator is not applicable to the Jason project (N/A). The resultant mutants are either equivalent to the original interpretation (E), non-equivalent and killed by the test set (K), or non-equivalent and *not* killed by the test set (NE).

**Test Assessment**

The non-equivalent and unkilled mutants indicate the weaknesses in the test sets. In order to kill such a mutant that the RSO operator leads to, we need a test that can capture the differences in the resultant agent behaviour between selecting all applicable plans and selecting only the first applicable plan. These plans must have the same triggering event, the contexts that are not mutually exclusive and the ability to affect the agent behaviour. In the DR project, the only two such plans are the plan to get beer when the fridge is empty (p1) and the plan to get beer when the owner reaches the limit of drinking (p2). Therefore, we can design a test on which the limit of drinking is just reached when there is no beer in the fridge by e.g., modifying the initial amount of beer in the fridge. This test will cause p2 to be executed twice under the mutated interpreter so that the owner will be advised about drinking twice.

In order to kill the non-equivalent mutant that the RES2 operator leads to, we need a test that can capture the differences in the resultant agent behaviour between interleaved execution of plans and non-interleaved execution of plans. These plans must have a chance to compete for execution and the ability to affect the agent behaviour. In the DR project, the only two such plans are the plan to move to the fridge and the plan to notify the current time (as requested by the owner on occasion). Therefore, we can design a test that can detect the difference in the agent behaviour – the robot under the original interpreter has a chance to notify the current time while moving to the fridge, while it always notifies the time after arriving at the fridge under the mutated interpreter. It is worth noting that since the robot takes much longer to stay at the

fridge (a few seconds) than to move to the fridge (less than one second) on the original test set, the agent has a much bigger chance to notify the time at the fridge than on the move although under the original interpreter. Therefore, we can increase the chance to notify the time on the move by e.g., largely increasing the map size (so that the robot will take longer to move), to make it more likely we will kill the mutant.

In order to kill the non-equivalent mutant that the BAP operator leads to, we need a test that can capture the differences in the resultant agent behaviour between different orderings of beliefs. In the GM II project, there is only one description that causes the order of beliefs to matter – the actions to announce to other teammates all gold deposits that the gold miner agent perceived and that have not been handled or announced yet. Under the original interpreter, the gold miner agent will first announce the gold it perceived most recently; under the mutated interpreter, it will first announce the gold it perceived initially. The different orders of gold announcements may cause other teammates to bid for and be allocated different gold. Therefore, we can add a test that can detect this difference. It is worth noting that this difference to the original behaviour may not be a violation of the correctness requirements; instead it may be just a tiny variation that reflects the non-determinism of multi-agent systems, in which case the mutant is considered as equivalent.

**Robustness Assessment**

Where our operators produced equivalent mutants, it indicates that the Jason project is robust to the corresponding semantic changes. From these equivalent mutants we can come up with some ideas of how to resist these changes. For instance, in order to resist the semantic changes caused by the BQO and GQO operators while not affecting the agent behaviour under the original interpreter, the agent description has to be improved so that there can be only one matched belief or goal at most for each query. To resist the semantic change caused by the GAP operator, the agent description can be improved so that the agent behaviour is independent of the order of the goal type events and intentions.

Those mutants that are or can be killed indicate that the Jason project is not robust to the corresponding semantic changes. For instance, the DR project does not behave correctly under the semantic change caused by the RSO operator. In order to be robust to this change the agent description can be improved so that there can be only one applicable plan at most in any case. As another example, the DR project does not behave correctly under the semantic change caused by the RES2 operator. In order to be robust to this change the agent description can be improved so that there can be only one non-empty competitive intention at most in any case. Another example is that the GM II project cannot resist the semantic change caused by the BAP operator. In order to be robust to this change the agent description can be improved so that the agent behaviour is independent of the order of the beliefs.

## 6 Conclusions

Semantic mutation testing (SMT) is a useful technique for assessing tests and the robustness of a program to semantic changes. In this paper we applied SMT to three agent languages, namely Jason, GOAL and 2APL. We showed that SMT for these languages is useful in several contexts – migration between languages, evolution of languages, common misunderstandings, and ambiguity of informal semantics. We derived sets of semantic mutation operators for these languages, and a broader set of semantic mutation operator classes that are applicable to rule-based agent languages. Finally, we used two Jason projects in a preliminary evaluation of the semantic mutation operators for Jason. The results suggest that SMT can indicate some weaknesses in test sets and programs.

Our future work will focus on further evaluation of the semantic mutation operators for Jason. To further evaluate the ability of these operators to assess tests, we will examine their *representativeness* in comparison to realistic misunderstandings and their *power* by looking for more hard-to-kill mutants (as we have done in this paper), as suggested by [8]. To further evaluate the ability of these operators to assess program robustness, we will apply them to more Jason projects and provide specific rules to change the agent description in order to improve robustness.

## References


1. Adra, S.F., McMinn, P.: Mutation operators for agent-based models. In: Proceedings of 5th International Workshop on Mutation Analysis. IEEE Computer Society (2010)
2. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
3. Bordini, R.H., Hübner, J.F.: Semantics for the Jason variant of AgentSpeak (plan failure and some internal actions). In: Proceedings of ECAI'10, pp. 635–640 (2010)
4. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)
5. Clark, J.A., Dan, H., Hierons, R.M.: Semantic Mutation Testing. Science of Computer Programming (2011)
6. Dastani M.: 2APL: A practical agent programming language. Autonomous Agents and Multi-Agent Systems 16(3), 214–248 (2008)
7. Dastani, M., van Riemsdijk, M.B., Meyer, J.J.C.: Programming multi-agent systems in 3APL. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Platforms and Applications, pp. 39–67. Springer, Heidelberg (2005)
8. Hindriks, K.V.: Programming rational agents in GOAL. In: Bordini R.H., Dastani M., Dix J., El Fallah Seghrouchni A. (eds.), Multi-agent programming: Languages, platforms and applications, vol. 2, pp. 3–37. Springer, Heidelberg (2009)
9. Houhamdi, Z.: Multi-agent system testing: A survey. International Journal of Advanced Computer Science and Applications (IJACSA) 2(6), 135–141 (2011)
10. Huang Z., Alexander R., Clark J.A.: Mutation Testing for Jason Agents. In: Dalpiaz F., Dix J., van Riemsdijk, M.B. (eds.) EMAS 2014. LNCS (LNAI), vol. 8758, pp. 309–327. Springer, Heidelberg (2014)

11. Jia Y., Harman M.: Higher order mutation testing. J Informat Softw Technol 51(10), pp. 1379–1393 (2009)
12. Mathur, A.P.: Foundations of Software Testing. Pearson (2008)
13. Nguyen, C.D., Perini, A., Bernon, C., Pavón, J., Thangarajah, J.: Testing in multi-agent systems. In: Gleizes, M.-P., Gomez-Sanz, J.J. (eds.) AOSE 2009. LNCS, vol. 6038, pp. 180–190. Springer, Heidelberg (2011)
14. Saifan, A.A., Wahsheh, H.A.: Mutation operators for JADE mobile agent systems. In: Proceedings of the 3rd International Conference on Information and Communication Systems, ICICS (2012)
15. Savarimuthu, S., Winikoff, M.: Mutation operators for cognitive agent programs. In: Proceedings of the 2013 International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2013), pp. 1137–1138 (2013)
16. Savarimuthu, S., Winikoff, M.: Mutation Operators for the GOAL Agent Language. In: Cossentino M., El Fallah Seghrouchni, A., Winikoff, M. (eds.) EMAS 2013. LNCS (LNAI), vol. 8245, pp. 255–273. Springer, Heidelberg (2013)
17. Tiryaki A.M., Oztuna S., Dikenelli O., Erdur R.C.: Sunit: A unit testing framework for test driven development of multi-agent systems. In: Agent-Oriented Software Engineering VII. LNCS, vol. 4405, pp. 156–173. Springer, Heidelberg (2006)
18. Winikoff M.: Novice programmers' faults & failures in GOAL programs. In: Proceedings of the 2014 International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2014), pp. 301–308 (2014)